REPAST JAVA GETTING STARTED

NICK COLLIER & MICHAEL NORTH - REPAST DEVELOPMENT TEAM

0. Before we Get Started

Before we can do anything with Repast Simphony, we need to make sure that we have a proper installation of the latest version. Instructions on downloading and installing Repast Simphony on various platforms can be found on the Repast website.[1] Repast Simphony requires Java 8 to be installed. Java can be found at the Java Standard Edition Downloads Page.[2]

1. Getting Started with Repast Simphony for Java

We will be building a simple agent-based model involving zombies chasing humans and humans running away from zombies. When we are finished, the running model should look like fig. 1.

The first thing we need to do is to turn on the Repast Simphony Perspective in the Eclipse IDE that comes with Repast Simphony. A perspective adds various menus, toolbar buttons and so forth to Eclipse to make it easier to develop certain kinds of projects. The Repast Simphony perspective does this for Repast Java development. To turn on the Repast Simphony perspective, click on the Window menu, then Open Perspective, Other. Choose the Repast Simphony perspective from the dialog that pops up (Fig. 2) and click OK. You should see the Repast Simphony perspective (Fig. 3) selected in the upper right hand corner of Eclipse. You should only have to select the perspective from the dialog once. When Repast Simphony is restarted the last perspective will be used and the last few will be available in the upper right hand corner.

With the perspective now enabled, we now need to create a new Repast Simphony project. Assuming you've started Repast Simphony, right click in the Package Explorer pane and choose "New" and then "Repast Simphony Project". This brings up the New Repast Simphony Project Wizard which gives us the ability to name our project (and a few more options which we'll ignore for now). Type jzombies in the "Project name" field, and press the "Finish" button. You should now see a jzombies project in the Package Explorer.

By default Repast Simphony will hide many of the details of the project. This is appropriate for ReLogo projects but not for those written in Java. If the ReLogo filters have not been previously disabled then we need to do that now. If you click on the triangle next to "jzombies" and you see a variety of folders (batch, docs, etc), then the filter has been

[1] https://repast.github.io/download.html
[2] http://www.oracle.com/technetwork/java/javase/downloads/index.html

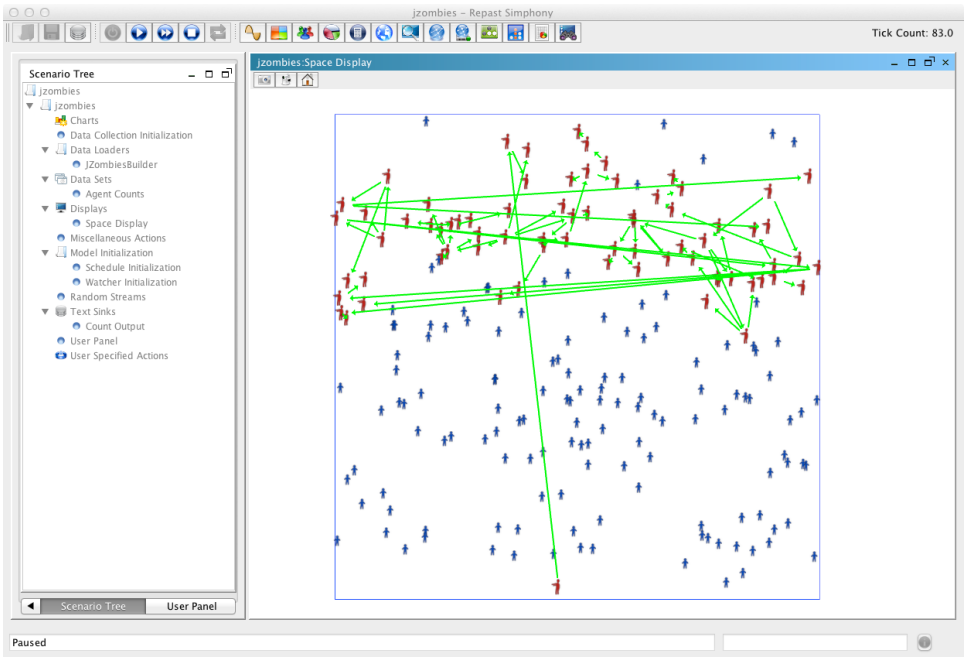

FIGURE 1. Completed Zombies Model

disabled. If you only see the src directory then the filter needs to be disabled. To disable the filter, click the downward pointing arrow in the "Package Explorer" pane. If you see "ReLogo Resource Filter" then click on it to disable the filter (Fig. 4). Otherwise, click on the "Filters" item. This brings up the Java Element Filters window. Scroll through the elements and click off the checkbox for ReLogo Resource Filter (Fig. 5).

1.1. **Building the Model.** In a Repast Simphony Java simulation, agents do not need to extend any base class or interface. They can be created simply by using Eclipse's "New Class Wizard". The Repast Simphony project wizard creates a source directory and default package into which we can create these agent classes. In our case, the package is "jzombies", which can be seen immediately under the src directory.[3]

We will now create our `Zombie` and `Human` classes.

(1) Right click on the jzombies folder under src
(2) Select New then Class from the Menu (Fig. 6)
(3) Type Zombie for the name

[3]Package names typically use an internet domain name as the basis for a package name, but for the purposes of this tutorial "jzombies" is perfectly fine. See Java Tutorial: packages for more info.

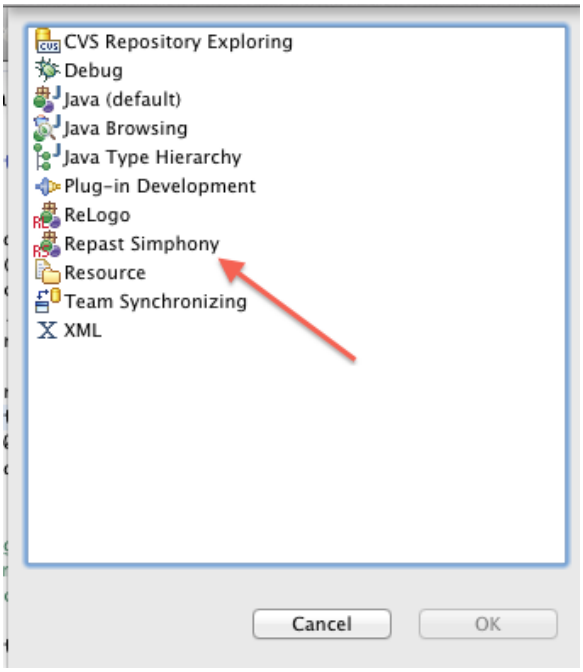

FIGURE 2. Perspective Dialog

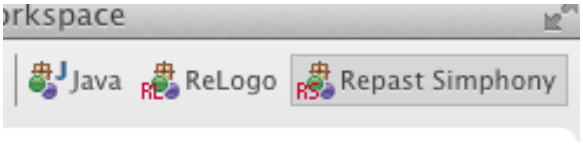

FIGURE 3. Repast Simphony Perspective Icon.

(4) Optionally, click the generate comments box. This won't comment the code for you, of course, but it does leave a placeholder.
(5) Repeat steps 1-4, but type Human for the name of the class

You should now see `Zombie.java` and `Human.java` files in the jzombies packages underneath the src folder, and these files should be open in Eclipse's editor pane. (If they are not open, double click on each file to open it). Let's begin with the Zombies. The Zombies behavior is to wander around looking for Humans to infect. More specifically, each iteration of the simulation, each Zombie will determine where the most Humans are

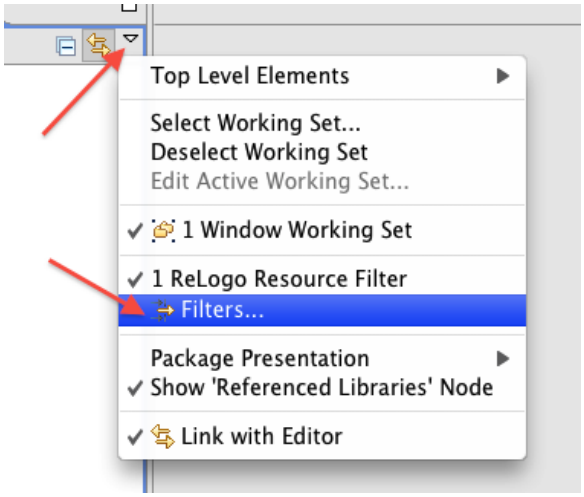

FIGURE 4. Disabling the ReLogo Filter.

within its local area and move there. Once there it will attempt to infect a Human at that location and turn it into a Zombie.

Rather than implementing that all at once, we will begin by implementing the movement. To do this we will locate the Zombies and Humans within a `ContinuousSpace` and a `Grid`. A `ContinuousSpace` allows us to use floating point numbers (e.g. 1.5) as the coordinates of a Zombie's and Human's location, and the `Grid` allows us to do neighborhood and proximity queries (i.e. "who is near me?") using discrete integer Grid coordinates.[4] Let's begin by adding some field variables and a constructor to the Zombie class. **Note that copying the text straight from this document may lead to irregular spacing and other issues in the Eclipse editor.**

[4]More details on `ContinuousSpace` and `Grid` can found in the Repast Java API documentation and the Repast Reference.

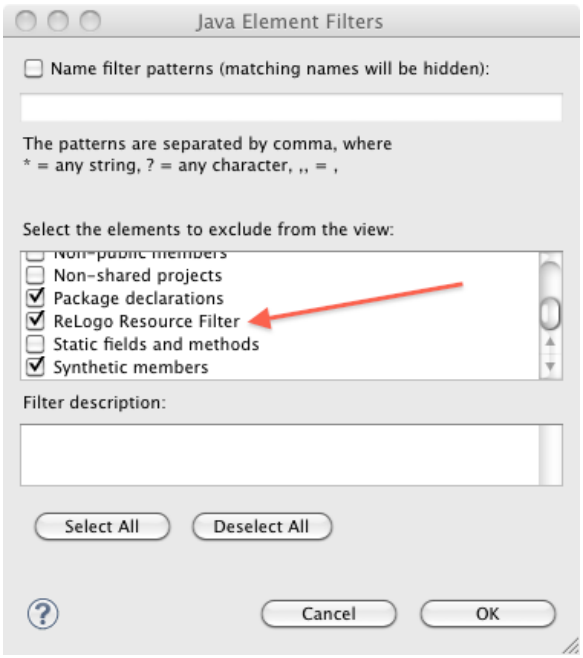

FIGURE 5. Filter Window.

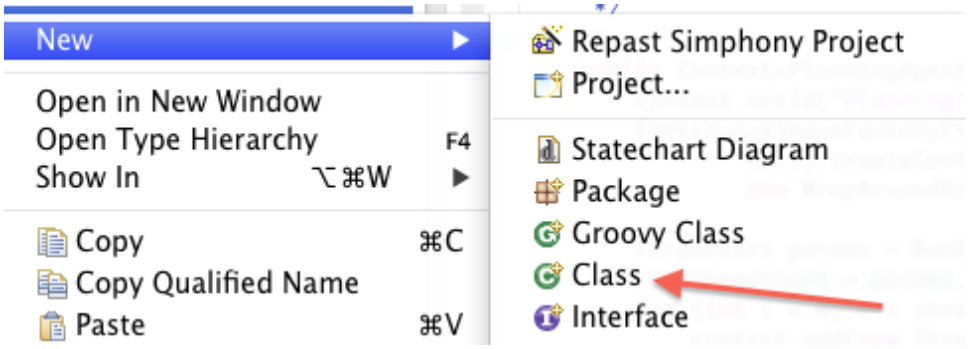

FIGURE 6. New Class Wizard Entry.

```
1  public class Zombie {
2
3     private ContinuousSpace<Object> space;
4     private Grid<Object> grid;
5
6     public Zombie(ContinuousSpace<Object> space, Grid<Object> grid) {
7        this.space = space;
8        this.grid = grid;
9     }
10 }
```

LISTING 1. Zombie Constructor and Variables

Line 3 and 4 are field variables to hold the space and grid in which the zombie will be located. The Zombie will move about the ContinuousSpace and we will simply round the ContinuousSpace location to determine the corresponding Grid location. Lines 6-9 constitute the constructor which sets the values of the `space` and `grid` variables. The `space` and `grid` variables have `Object` as their template parameter. This allows us to put anything in them and prevents Eclipse from giving us spurious warnings. You may need to add the appropriate imports for `ContinuousSpace` and `Grid`. These imports look like:

```
1 import repast.simphony.space.continuous.ContinuousSpace;
2 import repast.simphony.space.grid.Grid;
```

LISTING 2. Zombie Imports

You can add these automatically in Eclipse by right clicking on the type (e.g `Grid`) that needs an import and choosing Source - Add Import from the menu. You can also do cmd-shift-o (on OSX) or ctrl-shirt-o (Windows and Linux) to add the imports automatically. **As new code is added during the tutorial, you will need to add the appropriate imports. Whenever you see the red squiggle, use Eclipse to add the necessary imports.**

Now lets add a method to Zombie that will be called every iteration of the simulation. We will call this method `step`.

```
1  public void step() {
2     // get the grid location of this Zombie
3     GridPoint pt = grid.getLocation(this);
4
5     // use the GridCellNgh class to create GridCells for
6     // the surrounding neighborhood.
7     GridCellNgh<Human> nghCreator = new GridCellNgh<Human>(grid, pt,
8        Human.class, 1, 1);
9     // import repast.simphony.query.space.grid.GridCell
10    List<GridCell<Human>> gridCells = nghCreator.getNeighborhood(true);
11    SimUtilities.shuffle(gridCells, RandomHelper.getUniform());
12
13    GridPoint pointWithMostHumans = null;
14    int maxCount = -1;
15    for (GridCell<Human> cell : gridCells) {
16       if (cell.size() > maxCount) {
17          pointWithMostHumans = cell.getPoint();
18          maxCount = cell.size();
19       }
20    }
21 }
```

LISTING 3. step Method

Line 3 gets the location of this Zombie (`this` refers to the object whose method is calling the code) in the grid. We then use a `GridCellNgh` to create a `List` of `GridCells` containing Humans. A `GridCellNgh` is used to retreive a list of `GridCells` that represent the contents and location of the 8 neighboring cells around a `GridPoint`. The constructor to `GridCellNgh` takes the grid whose cells we want to get, the point whose neighboring grid cells we want, the Class of the items we want in the cells, and the extent along the appropriate dimensions. Lines 7 and 8 specify the grid and the location of this Zombie as the point whose neighboring cells we want. By specifying the Human class in the constructor and as a template parameter we filter out any non-Humans. Consequently, our GridCells will contain only Humans. We then call `getNeighborhood` passing `true` to it. By passing `true` the returned list of GridCells will contain the center cell where the Zombie is currently located. Using `SimUtilities.shuffle`, we shuffle the list of `GridCells`. Without the shuffle, the Zombies will always move in the same direction when all cells are equal. We use the `RandomHelper` class to provide us with a random generator for the shuffle. The remaining code iterates through the list of `GridCells` and determines which one of them has the largest size. A `GridCell`'s size is a measure of the number of objects it contains and thus the cell with the greatest size contains the most Humans.

Now that we have discovered the location with the most Humans (i.e. `pointWithMostHumans`), we want to move the Zombie towards that location. We will first write this method then add the code to call it in the `step()` method.

```
1  public void moveTowards(GridPoint pt) {
2     // only move if we are not already in this grid location
3     if (!pt.equals(grid.getLocation(this))) {
4        NdPoint myPoint = space.getLocation(this);
5        NdPoint otherPoint = new NdPoint(pt.getX(), pt.getY());
6        double angle = SpatialMath.calcAngleFor2DMovement(space,
7           myPoint, otherPoint);
8        space.moveByVector(this, 1, angle, 0);
9        myPoint = space.getLocation(this);
10       grid.moveTo(this, (int)myPoint.getX(), (int)myPoint.getY());
11    }
12 }
```

LISTING 4. moveTowards Method

`moveTowards()` begins with a check to make sure that the Zombie is not already at the location we want to move towards. Then we get the Zombie's current location in space as a `NdPoint`. `NdPoint` stores its coordinates as doubles and thus it is appropriate when working with `ContinuousSpace`s. We want the Zombie to move towards the `GridPoint pt`, but in order to make this sort of movement using a `ContinuousSpace` we need to convert that `GridPoint` to a `NdPoint`. Line 5 does that. We then use the `SpatialMath` method `calcAngleFor2DMovement` to calculate the angle along which the Zombie should move if it is to move towards the `GridPoint`. Line 8 then performs this movement and moves the Zombie in the `ContinuousSpace` 1 units along the calculated angle. The last two lines update the Zombies position in the `Grid` by converting its location in the `ContinuousSpace` to `int` coordinates appropriate for a `Grid`.

Once you've written this `moveTowards()` method, add a call to it in the `step()` method. The end of the `step()` should now look like:

```
1 for (GridCell<Human> cell : gridCells) {
2    if (cell.size() > maxCount) {
3       pointWithMostHumans = cell.getPoint();
4       maxCount = cell.size();
5    }
6 }
7 moveTowards(pointWithMostHumans);
```

LISTING 5. Step with MoveTowards Added

Note the addition of the call to `moveTowards()` following the for loop.

We want the `step` method to be called every iteration of the simulation. We can do this by adding an `@ScheduledMethod` annotation on it. Obviously, the method itself is part of a class, and thus what we are actually scheduling are invocations of this method on instances of this class. For example, if we have 10 Zombies, then we can schedule this method to be called on each of those Zombies. The annotation has a variety of parameters that are used to specify when and how often the method will be called, the number of object instances to invoke the method on, and so on. [5]

```
1 @ScheduledMethod(start = 1, interval = 1)
2 public void step() {
3    // get the grid location of this Zombie
4    GridPoint pt = grid.getLocation(this);
```

LISTING 6. Step Method with Annotation

The parameters here in line 1 will schedule the step method to be called on all Zombies starting at tick (timestep) 1 and every tick thereafter.

Let's now turn to the code for the Humans. Select Human.java in Eclipse's editor pane. The basic behavior for a Human is to react when a Zombie comes within its local neighborhood by running away from the area with the most Zombies. Additionally, Humans have a certain amount of energy that is expended in running away. If this energy is 0 or less then a Human is unable to run. We begin by creating the relevant field variables and constructor.

[5] See the Repast Java API documentation and the Repast Reference for more information on the `@ScheduledMethod` annotation.

```
1  public class Human {
2
3     private ContinuousSpace<Object> space;
4     private Grid<Object> grid;
5     private int energy, startingEnergy;
6     public Human(ContinuousSpace<Object> space, Grid<Object> grid,
7        int energy)
8     {
9        this.space = space;
10       this.grid = grid;
11       this.energy = startingEnergy = energy;
12    }
```

Listing 7. Human Constructor and Variables

The Human constructor is identical to that of the Zombie with the addition of `int energy` and `startingEnergy`. `energy` will be used to track the current amount of energy a Human has. `startingEnergy` will be used to set the energy level back to its starting level after a Human has had a rest.

The main behavior of a Human is implemented in its `run` method.

```
1  public void run() {
2     // get the grid location of this Human
3     GridPoint pt = grid.getLocation(this);
4     // use the GridCellNgh class to create GridCells for
5     // the surrounding neighborhood.
6     GridCellNgh<Zombie> nghCreator = new GridCellNgh<Zombie>(grid, pt,
7           Zombie.class, 1, 1);
8     List<GridCell<Zombie>> gridCells = nghCreator.getNeighborhood(true);
9     SimUtilities.shuffle(gridCells, RandomHelper.getUniform());
10
11    GridPoint pointWithLeastZombies = null;
12    int minCount = Integer.MAX_VALUE;
13    for (GridCell<Zombie> cell : gridCells) {
14       if (cell.size() < minCount) {
15          pointWithLeastZombies = cell.getPoint();
16          minCount = cell.size();
17       }
18    }
19
20    if (energy > 0) {
21       moveTowards(pointWithLeastZombies);
22    } else {
23       energy = startingEnergy;
24    }
25 }
```

Listing 8. The Run Method

This looks much like the Zombie code. A `GridCellNgh` is used to find the Zombies in the neighboring grid cells. It then determines which of these cells has the least Zombies and attempts to move towards that. Note that `moveTowards` is only called if the energy level is greater than 0. If energy does equal 0, the Human doesn't move and energy is set back to its starting level. A Human's `moveTowards` looks like:

```
1  public void moveTowards(GridPoint pt) {
2     // only move if we are not already in this grid location
3     if (!pt.equals(grid.getLocation(this))) {
4        NdPoint myPoint = space.getLocation(this);
5        NdPoint otherPoint = new NdPoint(pt.getX(), pt.getY());
6        double angle = SpatialMath.calcAngleFor2DMovement(space, myPoint,
7           otherPoint);
8        space.moveByVector(this, 2, angle, 0);
9        myPoint = space.getLocation(this);
10       grid.moveTo(this, (int)myPoint.getX(), (int)myPoint.getY());
11       energy--;
12    }
13 }
```

This is identical[6] to the Zombie's except that energy is decremented and the human moves 2 units rather than 1.

Unlike the Zombie code we are not going to schedule the `run()` method for execution. Rather we are going to setup a *watcher* that will trigger this run() method whenever a Zombie moves into a Human's neighborhood. We do this using the `@Watch` annotation. The `@Watch` annotation requires the class name of the class to watch, as well as a field within that class. The watcher will trigger whenever this field has been accessed. We can also define a query on the `@Watch` to further specify when the watcher will trigger.[7] Our `@Watch` annotation on `run()` looks like:

```
1  @Watch(watcheeClassName = "jzombies.Zombie",
2     watcheeFieldNames = "moved",
3     query = "within_moore 1",
4     whenToTrigger = WatcherTriggerSchedule.IMMEDIATE)
5  public void run() {
```

LISTING 9. Run with Watcher

This `Watch` will watch for any changes to a "moved" variable in the Zombies class. What this means is whenever any Zombie moves and their moved variable is updated, then this `Watch` will be checked for each Human. If the query returns true for that particular Human then `run` will be called immediately on that Human. Our query will return true when the

[6]We could factor out the duplicate code here to a Zombie and Human shared base class, but for the purposes of this tutorial it is easier to see the code in the class itself

[7]See the Repast Java API documentation and the Repast Reference for more details on the `@Watch` annotation.

Zombie that moved is within the Moore neighborhood (8 surrounding grid cells) of the Human whose `Watch` is currently being evaluated.

Sharp-eyed readers will note that we have not defined a "moved" variable in the Zombie class. So we will do that now. Click on the Zombie editor pane and add `private boolean moved;` below `private Grid<Object> grid;`. Then add `moved = true` to the `moveTowards` method.

```
1  public void moveTowards(GridPoint pt) {
2     // only move if we are not already in this grid location
3     if (!pt.equals(grid.getLocation(this))) {
4        NdPoint myPoint = space.getLocation(this);
5        NdPoint otherPoint = new NdPoint(pt.getX(), pt.getY());
6        double angle = SpatialMath.calcAngleFor2DMovement(space,
7           myPoint, otherPoint);
8        space.moveByVector(this, 1, angle, 0);
9        myPoint = space.getLocation(this);
10       grid.moveTo(this, (int)myPoint.getX(), (int)myPoint.getY());
11
12       moved = true;
13    }
14 }
```

Listing 10. Zombie MovedTowards with Moved

That completes this part of the code for Zombies and Humans. Now we need to turn to initializing the simulation. In a Java Repast Simphony model, initialization takes place in a class that extends the Repast Simphony class `ContextBuilder`. So, let's create one of these now.

(1) In the Package Explorer pane, click on the "jzombies" package underneath the src directory.
(2) Select New then Class from the Menu (Fig. 6)
(3) In the Name field, type JZombiesBuilder
(4) Click the Add button next to the interfaces field
(5) Under Choose interfaces type ContextBuilder and click OK
(6) Click Finish

An editor pane for `JZombiesBuilder.java` should now be open. You will notice a red squiggle under the "T" in `ContextBuilder<T>` and under `JZombiesBuilder` in the class declaration. Replace the "T" with Object to fix the former issue. The error on `JZombiesBuilder` is because `JZombiesBuilder` does not implement methods required by the `ContextBuilder` interface. We can use Eclipse to automatically implement the required

methods. Right click on `JZombiesBuilder`, choose Source then Override / Implement Methods. Click OK on the dialog box that pops up. (Fig. 7).

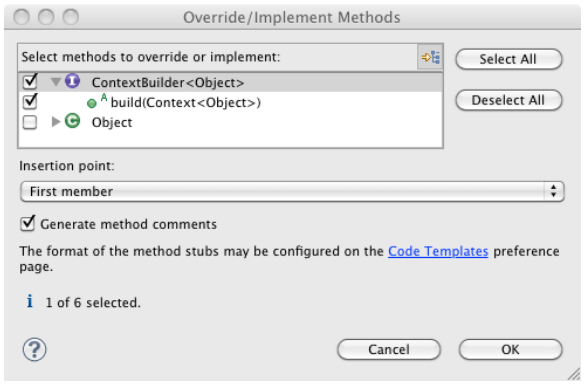

FIGURE 7. Override / Implement Methods Dialog.

```
1  public class JZombiesBuilder implements ContextBuilder<Object> {
2
3      /* (non-Javadoc)
4       * @see repast.simphony.dataLoader.ContextBuilder
5         #build(repast.simphony.context.Context)
6       */
7      @Override
8      public Context build(Context<Object> context) {
9          // TODO Auto-generated method stub
10         return null;
11     }
12 }
```

LISTING 11. Skeleton JZombiesBuilder

Your `JZombiesBuilder` should now look like the above. A ContextBuilder does just what it says it builds a `Context`. A `Context` is essentially a named set of agents. Building one consists of naming it, and adding agents to it. Contexts can also have `Projection`s associated with them. A `Projection` takes the agents in a `Context` and imposes some sort of structure on them. Our `ContinuousSpace` and `Grid` are projections. They take agents and locate them in a continuous space and matrix like grid respectively. In `JZombiesBuilder.build` we are thus going to create the agents and our `ContinuousSpace` and `Grid` projections. Let's begin by initializing the `Context` and creating the projections.

```
1  public Context build(Context<Object> context) {
2     context.setId("jzombies");
3
4     ContinuousSpaceFactory spaceFactory =
5        ContinuousSpaceFactoryFinder.createContinuousSpaceFactory(null);
6     ContinuousSpace<Object> space =
7        spaceFactory.createContinuousSpace("space", context,
8           new RandomCartesianAdder<Object>(),
9           new repast.simphony.space.continuous.WrapAroundBorders(),
10          50, 50);
11
12    GridFactory gridFactory = GridFactoryFinder.createGridFactory(null);
13    // Correct import:  import repast.simphony.space.grid.WrapAroundBorders;
14    Grid<Object> grid = gridFactory.createGrid("grid", context,
15            new GridBuilderParameters<Object>(new WrapAroundBorders(),
16            new SimpleGridAdder<Object>(),
17             true, 50, 50));
18
19    return context;
20 }
```

LISTING 12. JZombiesBuilder.build 1

Note that when doing adding the imports for this code, Eclipse may add the incorrect import for WrapAroundBorders. The correct import should be `import repast.simphony.space.grid.WrapAroundBorders;` not `import repast.simphony.space.continuous.WrapAroundBorders`.

We begin in line 1 by setting the id of the passed in `Context` to "jzombies". Typically, the id should be set to whatever the project name is and should match the context id in the context.xml file (more on this below). The remaining code creates the `ContinuousSpace` and `Grid` projections. In both cases, we begin by getting a factory of the appropriate type and then use that factory to create the actual `ContinuousSpace` or `Grid`. Both factories take similar parameters:

- the name of the grid or space
- the context to associate the grid or space with
- an `Adder` which determines where objects added to the grid or space will be initially located
- a class that describes the borders of the grid or space. Borders determine the behavior of the space or grid at its edges. For example, WrapAroundBorders will wrap the borders, turning the space or grid into a torus. Other border types such as `StrictBorders` will enforce the border as a boundary across which agents cannot move.

- the dimensions of the grid (50 x 50 for example).

The `GridFactory` differs slightly in that it bundles the borders, adder, dimensions etc. into a GridBuilderParameters object. The `GridBuilderParameters` also takes a boolean value that determines whether more than one object is allowed to occupy a grid point location at a time. With this in mind then, the above code creates a `ContinuousSpace` named "space" and associates it with the passed in `Context`. Any object added to this space will be added at a random location via the `RandomCartesianAdder`. The borders of the space will wrap around forming a torus, set via the `repast.simphony.space.continuous.WrapAroundBorders()`. Lastly, the dimensions of the space will be 50 x 50. This code also creates a `Grid` called "grid" and associates it with the `Context`. The grid will also wrap and form a torus. Objects added to this grid will be added with the `SimpleGridAdder` which means that they are not given a location when added, but rather held in a kind of "parking lot" waiting to be manually added via one of the `Grid`'s methods. The `true` value specifies that multiple occupancy of a grid location is allowed. The grid's dimensions will be 50 x 50. Note the we are using the `SimpleGridAdder` here so that we can manually set an agent's `Grid` location to correspond to its `ContinuousSpace` location. We will do this later in the `build` method.

Now let's create the agents. Add the following code prior to the return statement.

```
1  int zombieCount = 5;
2  for (int i = 0; i < zombieCount; i++) {
3     context.add(new Zombie(space, grid));
4  }
5
6  int humanCount = 100;
7  for (int i = 0; i < humanCount; i++) {
8     int energy = RandomHelper.nextIntFromTo(4, 10);
9     context.add(new Human(space, grid, energy));
10 }
```

LISTING 13. JZombiesBuilder.build 2

This should be relatively straight forward. We create a specified number of Zombies and Humans by looping through some creation code the specified number of times. We add the new Zombies and Humans to context. In adding them to the context we automatically add them to any projections associated with that context. So in this case, the Zombies and Humans are added to the space and grid using their `Adders` as described above. The Humans are created with a random energy level from 4 to 10. We use the `RandomHelper` to do this for us. In general, all random number type operations should be done through the RandomHelper.[8]

[8]See RandomHelper in Repast Java API documentation for more information.

Lastly, we will add the code to move the agents to the `Grid` location that corresponds to their `ContinuousSpace` location. The entire `build` method with this code added follows.

```
1  public Context build(Context<Object> context) {
2     context.setId("jzombies");
3     ContinuousSpaceFactory spaceFactory =
4     ContinuousSpaceFactoryFinder.createContinuousSpaceFactory(null);
5     ContinuousSpace<Object> space =
6     spaceFactory.createContinuousSpace("space", context,
7        new RandomCartesianAdder<Object>(),
8        new repast.simphony.space.continuous.WrapAroundBorders(),
9        50, 50);
10
11    GridFactory gridFactory = GridFactoryFinder.createGridFactory(null);
12    Grid<Object> grid = gridFactory.createGrid("grid", context,
13             new GridBuilderParameters<Object>(new WrapAroundBorders(),
14             new SimpleGridAdder<Object>(),
15             true, 50, 50));
16
17    int zombieCount = 5;
18    for (int i = 0; i < zombieCount; i++) {
19       context.add(new Zombie(space, grid));
20    }
21
22    int humanCount = 100;
23    for (int i = 0; i < humanCount; i++) {
24       int energy = RandomHelper.nextIntFromTo(4, 10);
25       context.add(new Human(space, grid, energy));
26    }
27
28    for (Object obj : context) {
29       NdPoint pt = space.getLocation(obj);
30       grid.moveTo(obj, (int)pt.getX(), (int)pt.getY());
31    }
32
33    return context;
34 }
```

LISTING 14. JZombiesBuilder.build Complete

The new code starting at line 28 simply iterates through the all agents in the context, retrieves each one's location in the `ContinuousSpace` and moves it to the corresponding location in the `Grid`.

Before we run this model, we need to update the metadata that the Repast Simphony runtime uses to help create displays and other runtime components. Open the jzombies.rs folder under the jzombies project folder, and double click on the `context.xml` file (fig. 8).

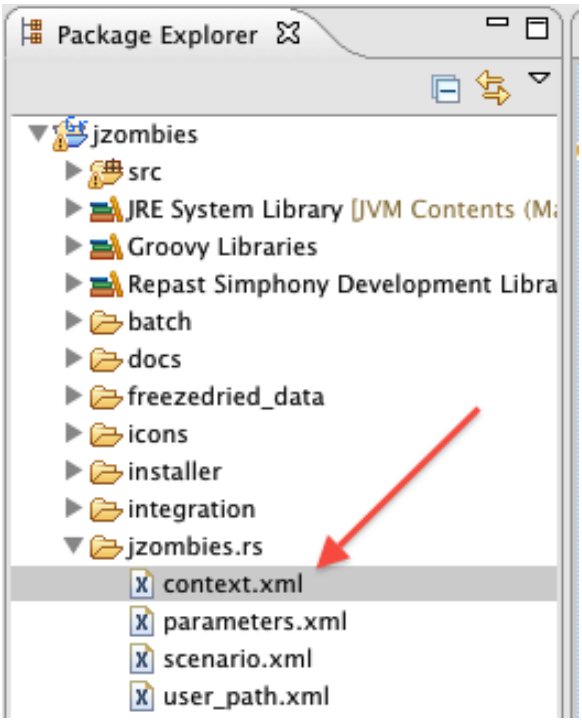

FIGURE 8. context.xml location

That should bring up the XML editor for editing the `context.xml` file. The `context.xml` file describes the context hierarchy for your model. The context hierarchy is composed of the contexts your model uses and the projections associated with them. Recall that in our context builder `JZombiesBuilder` we have a single context and created two projections. We need to update the context.xml to reflect this. The XML editor has two views associated with it, a Design view and a Source view (fig. 9). Click on the Source view. By default, the context.xml editor should have a context already in it with an id of "jzombies". (Recall that we set the id of our context to "jzombies".) You should see some xml that looks like the following listing.

```
1 <context id="jzombies"
2    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3    xsi:noNamespaceSchemaLocation="http://repast.org/scenario/context">
4
5 </context>
```

LISTING 15. context.xml 1

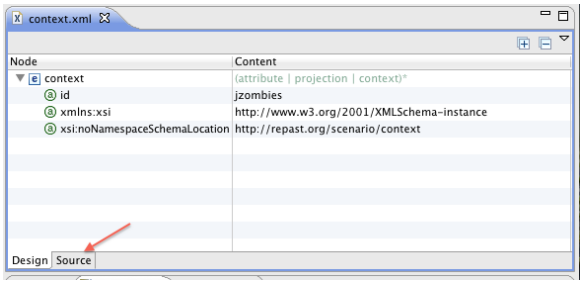

FIGURE 9. Context.xml editor

XML works in terms of elements and attributes. Elements are enclosed in angle brackets and attributes consist of a name followed by an equals sign followed by a quoted string. The quoted string is the value of the attribute. In the example above we have a context element which has an id attribute whose value is jzombies.

We need to add additional elements that describe the projections that our context contains. A projection element has two required attributes, an id and a type. The id is the name of the projection, e.g. the "grid" and "space" names we used in the projections in our ContextBuilder code. The type attribute is the type of the projection: a continuous space or grid in this case. Add the following to the context.xml by typing it into the context.xml source editor just above the `</context>`.

```
1 <projection type="continuous space" id="space"/>
2 <projection type="grid" id="grid"/>
```

LISTING 16. context.xml 2

Note that as you type eclipse will attempt to autocomplete for you. So, as you type an opening bracket '<' you may see a popup menu suggesting possible elements, one of which is a projection. Feel free to select that. Eclipse will also autocomplete the attributes for you with a default type of "network" and an empty id. You can position the cursor inside the attribute value (i.e between the quotes) and hit the command + space (on OSX) or

control + space (on Windows and Linux) keys to trigger the autocomplete menu. You can then choose from a menu of attribute values. Regardless of how you enter the XML, you should end up with the following:

```
1 <context id="jzombies"
2    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3    xsi:noNamespaceSchemaLocation="http://repast.org/scenario/context">
4
5    <projection type="continuous space" id="space"/>
6    <projection type="grid" id="grid"/>
7 </context>
```

LISTING 17. context.xml Complete

The order of the projection elements doesn't matter and the order of the attributes within the elements also doesn't matter. If you used the autocomplete, you may see your projection elements ending with something like `id="space"></projection>`. That is fine as well. Save the context.xml file.

Now its time to launch the model. When we created our project using the Repast Simphony Project Wizard, it automatically created Eclipse launchers for us, and we can use those to launch the model. If you click on the small downward facing triangle next to the Eclipse launcher button (fig. 10), you'll see the various available launchers. Click on "jzombies Model".

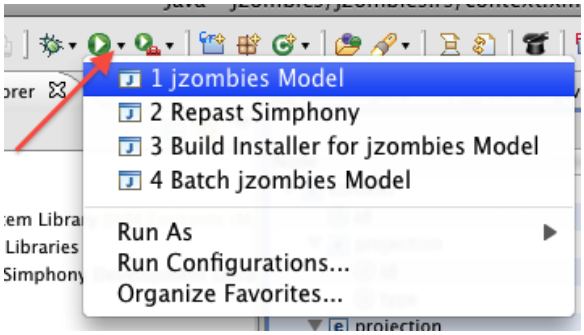

FIGURE 10. jzombies Model Launcher

This will start the Repast Simphony Runtime. Note that you may see some initial errors along the lines of `SpaceProjectionBuilder - Unable to build continuous space 'space' ...`. These can be ignored for now and in our next steps we are going to correct them. We need to do some runtime configuration before we run the model itself. We are going to setup the runtime to use our context builder to initialize the model and also create

an initial display. We setup the runtime to use our context builder by specifying the data loader.

(1) In the Scenario Tree, right click on the Data Loaders node and click "Set Data Loader". If the tree is not visible use the tab controls on the left side of the runtime frame to select the Scenario Tree.
(2) In the "Select Data Source Type" window, click on "Custom ContextBuilder Implementation". Click Next.
(3) You should see the name of our context builder class in the combo box, `jzombies.JZombiesBuilder`, if not, use the combo box to find it. If the box is empty, go back and check your code. This typically means that there was a compilation error. Click Next.
(4) Click Finish.

You should now see JZombiesBuilder as the name of the Data Loader (fig. 11)

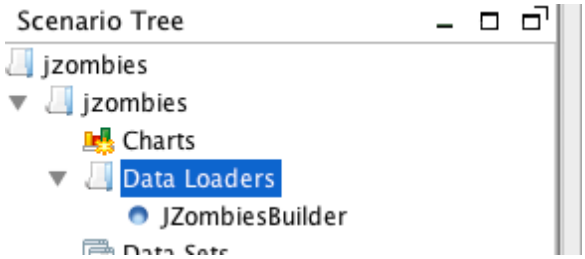

FIGURE 11. Data Loader

We will now create a simple display.

(1) Right click on Displays in the Scenario Tree and click "Add Display"
(2) In the Display configuration dialog, type Space Display for name. Leave 2D as the type.
(3) Select our "space" projection as the one we want to display. Click on space in the "Projection and Value Layers" section and then click the green arrow. The projections on the right are the ones will be displaying and those on the left are all the possible projections to display. The dialog should now look like fig. 12
(4) Click Next.
(5) Select the Human and Zombie agents as the types we want to display. Do this by selecting each in the left and then clicking the right pointing arrow to move them to right. If Zombie is not at the top of the list on the left use the up and down

arrows to move it to the top. By putting Zombie at the top, the visualized Zombies will be displayed over the Humans if they do overlap. The dialog should now look like fig. 13

(6) Click Next.
(7) In this panel, we can configure what we want the Zombies and Humans to look like. This can be done programmatically by specifying a class that implements `repast.simphony.visualizationOGL2D.StyleOGL2D` or via a wizard. We will use the wizard to create a simple style for our agents. Click the button to the right of the style class combo box (fig. 14).
(8) In the 2D Shape editor, change the Icon Shape to a square using the combo box and change the color to red by clicking the button with the blue square, and choosing a red color from the icon color dialog. Click OK on the icon color box, then OK on the 2D Shape Editor.
(9) Repeat the previous step for the Human. Click on Human in the list of Agents. Then click the icon editor button as before. Leave the default as is, and click OK.
(10) Click Next
(11) Click Next
(12) Click Finish

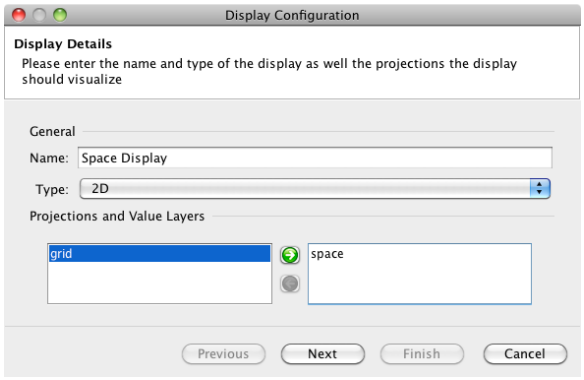

FIGURE 12. Configuring the Display

You should now see "Space Display" under the Display node in the Scenario Tree. Save your new scenario info (the new Data Loader and Display) by clicking the "Save" button (fig. 15) on the Repast Simphony runtime toolbar.

We can now run our model. Click the Initialize button (fig. 16) to initialize the model and bring up the display. If the display is not centered, you can center it by clicking on the display "home" button to reset the view. The mouse wheel will zoom the display in and out as will holding down the shift key and right mouse button and moving the mouse

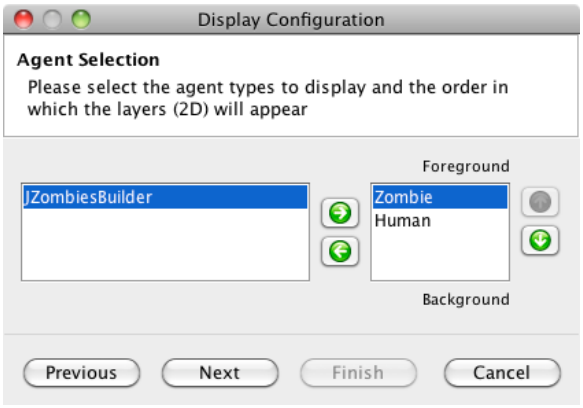

FIGURE 13. Configuring the Display 2

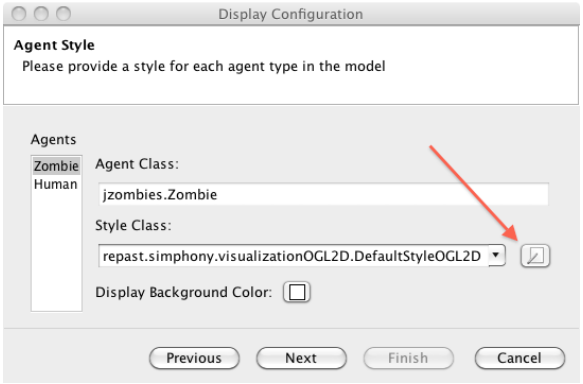

FIGURE 14. Configuring the Display 3

up and down. You can run the simulation by clicking the Run button, step through each timestep with the Step button, stop the simulation with the Stop button, and reset it for another run with the Reset button (fig. 16). When the simulation has been stopped, you must reset it with the Reset button in order do any more runs.

The display should show 5 red zombie squares and 100 blue human circles. If you repeatedly click the step button (fig. 16), you should see the zombies move around. When they do get near a human, the human will move away. Note that a zombie may become stuck in a crowd of humans. This location will be the most desirable place for a zombie and thus the zombie won't move. You can also click the play button to run the model

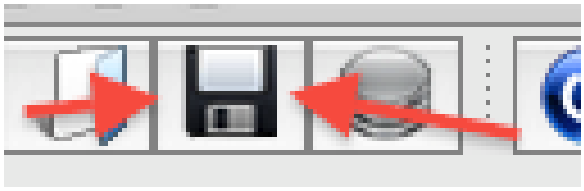

FIGURE 15. Save Scenario Button

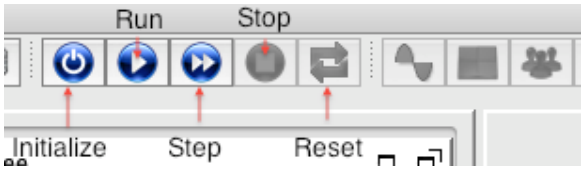

FIGURE 16. Repast Simphony Simulation Buttons

without stepping. If this model runs too fast, you can change the "tick delay" in the Run Options Panel tab.

Now let's turn back to the code and add some additional behavior. We will now make the Zombies "infect" the Humans and turn them into Zombies. We will also add a network projection to model the Zombie infection network. In Eclipse, open the `Zombie.java` file, if its not already open. If it is open, select it in the editor pane. Let's add a new method to the Zombie class called `infect`.

```
1  public void infect() {
2     GridPoint pt = grid.getLocation(this);
3     List<Object> humans = new ArrayList<Object>();
4     for (Object obj : grid.getObjectsAt(pt.getX(), pt.getY())) {
5        if (obj instanceof Human) {
6           humans.add(obj);
7        }
8     }
9     if (humans.size() > 0) {
10       int index = RandomHelper.nextIntFromTo(0, humans.size() - 1);
11       Object obj = humans.get(index);
12       NdPoint spacePt = space.getLocation(obj);
13       Context<Object> context = ContextUtils.getContext(obj);
14       context.remove(obj);
15       Zombie zombie = new Zombie(space, grid);
16       context.add(zombie);
17       space.moveTo(zombie, spacePt.getX(), spacePt.getY());
18       grid.moveTo(zombie, pt.getX(), pt.getY());
19
20       Network<Object> net = (Network<Object>)context.
21          getProjection("infection network");
22       net.addEdge(this, zombie);
23    }
24 }
```

LISTING 18. Infect Method

In summary, the `infect()` method gets all the Humans at the Zombie's grid location. A Human is chosen at random from these Humans. The chosen Human is removed from the simulation and replaced with a Zombie. More particularly, the `infect` method begins by getting the grid location of `this`, the Zombie calling this code. It then creates a `List` into which all the Humans at the grid location are put. The loop starting in line 4 iterates through all the `Objects` at that location, and if they are instances of the Human class it adds them to the list. If the size (the number of elements in the list) of the list is greater than 0 then there are Humans at this location, and we choose one at random. To choose one at random we draw a random number, using RandomHelper, to use as the index into the list. We then get the Object at that index in the list. We get the location in "space" of this random human and remove it from the Context. Using the `ContextUtils` class we get the `Context` that contains the random human. (Note we need an import for the repast.simphony.Context and it may be far down the list if you auto-import.) We then remove it from the context. Removing an object from the context automatically removes it from any projections associated with that context. So, in our case, the random human

is removed from the "space" and "grid" projections. Lastly, we create a new `Zombie` and add it to the context. Then in line 17, we move it to the random human's location in space using the "spacePt" that we retrieved in line 12. We set its grid location to the grid coordinates of this Zombie, using the "pt" that we retrieved in line 2. Lastly, we close by getting our infection network by name from the context. (We haven't created this network yet, but will do so shortly.) An edge between `this`, the Zombie that infected the Human, and `zombie` is then created.[9]

We also need to add a call to `infect` to the Zombie's `step` method. Add `infect()` immediately after `moveTowards(pointWithMostHumans)` in the `step` method.

The final step in the code is to create the network in our `JZombiesBuilder` just as we created the "space" and "grid" projections. Open `JZombiesBuilder.java` and add the following to top of the `build` method.

```
1 NetworkBuilder<Object> netBuilder = new NetworkBuilder<Object>
2    ("infection network", context, true);
3 netBuilder.buildNetwork();
```

We use a `NetworkBuilder` to create the network. This takes a name ("infection network"), the context to associate the network with, and whether or not the network is directed. In our case, we want a directed network with links coming from the infecting zombie to the zombie it created, and so the value is `true`.[10] In order to actually build the network we then call `buildNetwork`. This will create the network projection and automatically associate it with the context. Any agents added to the context will then become nodes in this network.

Recall that we had to add "space" and "grid" to the context.xml file. We now need to do the same for our network. The procedure is the same. Open the `context.xml` file in Eclipse. Add a new projection element to the context and set its id to "infection network" and its type to "network". The result should look like the following;

```
1 <context id="jzombies"
2    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3    xsi:noNamespaceSchemaLocation="http://repast.org/scenario/context">
4
5    <projection type="continuous space" id="space"/>
6    <projection type="grid" id="grid"/>
7    <projection type="network" id="infection network"/>
8 </context>
```

[9] See the Java API docs and the user manual for details on working with Network projections

[10] A `NetworkBuilder` can build networks from a variety of sources and configurations. See the Java API docs and user manual for details.

Now let's run our simulation. Start the jzombies Model using the Eclipse launcher as you did before (fig. 10). Once the Repast Simphony runtime has come up, initialize and run the simulation. You should now see humans becoming zombies. If you don't, then check your code for mistakes. Notice that the network is not displayed though. We need to update the display to show the network and in doing so we will also update the styles for the Humans and Zombies. Let's remove the old display first. Right click on the "Space Display" in the Scenario Tree under Displays and then "Delete". Now we will make the new display.

(1) Right click on Displays in the Scenario Tree and click "Add Display"
(2) In the Display configuration dialog, type Space Display for name. Leave 2D as the type.
(3) Select our "space" and "infection network" projections as the ones we want to display. Click on space in the "Projection and Value Layers" section and then click the right pointing green arrow. Repeat for the "infection network".
(4) Click Next.
(5) As before, select the Human and Zombie agents as the types we want to display. Do this by selecting each in the left and then clicking the right pointing arrow to move them to the right. If Zombie is not at the top of the list on the left, use the up and down arrows to move it to the top. The dialog should now look like fig. 13
(6) Click Next.
(7) In this panel, we can configure what we want the Zombies and Humans to look like. This can be done programmatically by specifying a class that implements `repast.simphony.visualizationOGL2D.StyleOGL2D` or via a wizard. We will use the wizard to select an icon for our agents. With Zombies selected in the list of Agents, click the button to right of the style class combo box (fig. 14).
(8) In the 2D Shape editor, click on the "Select Icon File" button. We want to use the zombie.png icon that comes with the jzombies demo model. Navigate to where the demo models are installed and click on zombies.png in the `jzombies/icon` directory. (If you can't find zombies.png, feel free to style the Zombie as a circle or whatever, using the 2D Shape editor).
(9) Click OK when you have finished.
(10) Repeat the previous step for the Human. Click on Human in the list of Agents. Then click the icon editor button as before. Click the"Select Icon File" button and navigate to the `jzombies/icon` directory in the demo model. Choose the person.png icon.
(11) Click OK when you have finished with the 2D Shape editor.
(12) Click Next
(13) Click Next
(14) Now you can style the infection network. Click the button to the right of the Edge Style Class combo box and then click the button with the black colored square on

it to change the edge's color. Click a reddish color and then OK to close the color dialog. Click OK to close the 2D Edge Style Editor.
(15) Click Next
(16) Click Finish

You should now see a "Space Display" entry under the "Display" node in the Scenario Tree. Click the "Scenario Save" button (fig. 15) to save your new scenario configuration.

Run the simulation and you should see the zombies, humans and the zombie infection network. Note that some edges may span the length of the space. Recall that the world is a torus and these edges are in fact between Zombies at the edge of the space. The edge itself can be though of as wrapping around one side and into the other.

1.1.1. *Data Collection.* Now let's add some data collection to the model. The data we want to record is the number of Zombies and number of Humans at each timestep. Repast Simphony records data from *Data Sources*. The wizard can be used to define Data Sources or you can create them yourself. Data Sources come in two flavors Aggregate and Non-Aggregate. Aggregate data sources receive a collection of objects (agents, for example) and typically return some aggregate value calculated over all the objects. For example, an aggregate data source might call a method on each object and return the maximum value. A non-aggregate data source takes a single object (e.g. a single agent) and returns a value. For example, a non-aggregate data source might call a method on an agent and return the result of that method call. Note that Repast Simphony will take care of which objects to pass to a data source and the actual data collection. You just need to define the data source(s).

Data collection is setup up in Repast Simphony by defining the data sources described above. These data sources are collected in data sets. Think of a data set as a template for producing tabular data where each column represents a data source and each row a value returned by that data source. To add a data set,

(1) Launch the jzombies Model (exit it and relaunch it if it is already running) via the Eclipse launch button.
(2) Right click on "Data Sets" in the Scenario Tree, and click "Add Data Set"
(3) In the Data Set Editor, type Agent Counts as the Data Set ID, and Aggregate as the Data Set type. Recall that we want to record the number of Zombies and Humans at each time step. Such counts are considered aggregate operations in that they operate on collections of agents.
(4) Click Next
(5) In this step, you specify the data sources. The Standard Sources tab allows you to add some standard data sources to your data set. Selecting the tick count check box will create a data source that returns the current tick, the Run Number will return the current run number and Random Seed the current random seed. If the tick count box is not selected, select it now.

(6) Click on the Method Data Source tab. The Method Data sources tab allows you to create data sources that will call a method on agents in your model and then perform some aggregate operation on the values returned by that method call.
(7) Click the Add button. You should see a row added to the method data sources table that looks like (fig. 17). Note that the row may not exactly match that of the figure.
(8) Double click on the empty table cell in the Source Name column and enter Human Count.
(9) If the Agent Type is not Human, double click on the Agent Type cell and select Human from the drop down list.
(10) If the Aggregate Operation is not Count, double click on the Aggregate Operation cell in the table and select Count. The aggregate operation determines what sort of operation is performed on the results of the method call. For example, Min, Max and Mean can be selected here. The Count operation returns the number of objects of the type specified in the Agent Type cell currently in the simulation. No method call is actually applicable here and so that should automatically be set to N/A.
(11) Repeat the previous steps beginning with step 7 to add a new row and edit that row to provide a Zombie count. The editor should now look like fig. 18. (The Custom Data Source panel allows you to enter the name of a class that implements either AggregateDataSource or NonAggregateDataSource. We are not using this in this Zombies model.)
(12) Click Next
(13) The Schedule Parameters panel allows you edit when the data will be recorded. We want to record our data after all the Zombies have moved. Recall that we scheduled the Zombie's `step` method to start at tick 1 and every tick thereafter. We want to record our data after this. Note that the 'Priority" here is Last which should insure that this records after all the Zombies have moved.
(14) Cick Finish

You should now see an "Agent Counts" node underneath the "Data Sets" node.

1.1.2. *Writing Data.* If we run the model now, data will be recorded but we have not set up anywhere for it to be written to. Simphony can write data to both a file and the console. Console output will show up in Eclipse's console tab and can be useful for debugging. File output will write the recorded data out to a file. We will create some file output by defining a File Sink.

(1) Right click on "Text Sinks" in the Scenario Tree, and click "Add File Sink".
(2) The File Data properties pane allows us to choose what data we want the file sink to write. Enter Count Output as the File Sink name. If we had created more than

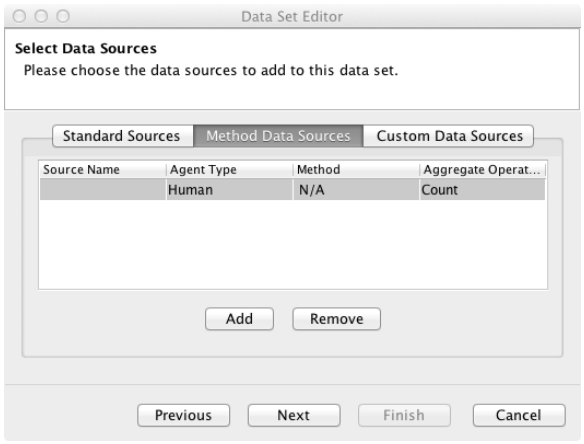

FIGURE 17. Selecting Data Sources

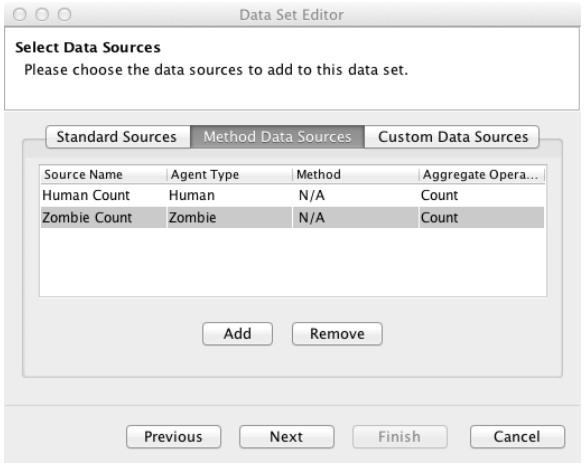

FIGURE 18. Selecting Data Sources 2

one data set we might have to choose it from the Data Set Id combo box, but as we didn't the default "Agent Counts" data set is fine. Beneath the Data Set ID box, you will see two columns. The first column lists the data sources associated with the currently selected data set. The second column displays the data sources that will be written to the file. Click on the tick entry in the source list and then click the right arrow button to move it over to the "to-be-written" list. Repeat for

the other data sources. You can use the up and down arrows to change the order of the data sources.
(3) Click Next
(4) The File Properties panel allows us to set some additional file properties: the file name and what meta data to put in the file name. We can leave the defaults as they are. The delimiter specifies what string to use to separate the data values. The format type can be tabular or line. Tabular is the typical CSV spreadsheet table format, while line will precede each data value with the data source name followed by a colon. In our case, this would look something like: tick: 1, Human Count: 20, Zombie Count: 5
(5) Cick Finish

Click the "Scenario Save" button (fig. 15) to save your new scenario configuration. We can now run the simulation and see the output. Initialize and run the simulation for a 100 or so ticks. By default the output will be written to your project directory. In Eclipse, right click on the root jzombies folder in the Package Explorer and choose "Refresh". You should now see a file with a name like `ModelOutput.2013.Apr.19.14_38_26_EST.2.1.txt` in your project directory. If you double click on it you can see the recorded data. (Note that I've justified the text for readability in the listing below.)

```
1 "tick",  "Human Count", "Zombie Count"
2 1.0,     199,           6
3 2.0,     199,           6
4 3.0,     198,           7
5 4.0,     195,           10
6 ...
```

Listing 19. Zombie Model Output

1.1.3. *Creating a Chart.* We can use the same data set to create a time series type chart that will plot the Human Zombie counts over time.

(1) If you have just completed a run, make sure to reset the simulation. Right click on the Charts entry in the Scenario Tree and select Add Time Series Chart.
(2) Enter Agent Counts Chart for the name and choose Agent Counts for the data set. It should be the default if the you have not added any additional data sets.
(3) The Chart Data Properties panel allows you to select the data sources to plot in the chart, as well as specify the legend label and color for each data source. Each selected data source will become a series in the chart. Click on the check boxes for the Human and Zombie Count data sources. The default labels and colors are fine (see fig. 19) but you can change them by double clicking on the label and color entries respectively.

(4) Click Next
(5) The Chart Properties panel allows you to set the chart's properties. Enter Agent Counts as the title and Number of Agents as the Y-Axis label. The default properties are fine but feel free to change the colors and experiment with the X-Axis range. A range of -1 means the entire range is shown in the x-axis. Anything else restricts the x-axis to displaying the specified range.
(6) Click Finish.

A new Agent Counts Chart should now appear as entry under the Charts tree node. Click the save scenario button to save the chart definition to the scenario. Run the simulation and you should now see a chart displaying the Zombie and Human counts over time. To see the chart click on the "Agent Counts Chart" tab in the display panels (on the bottom tab next to the Space Display).

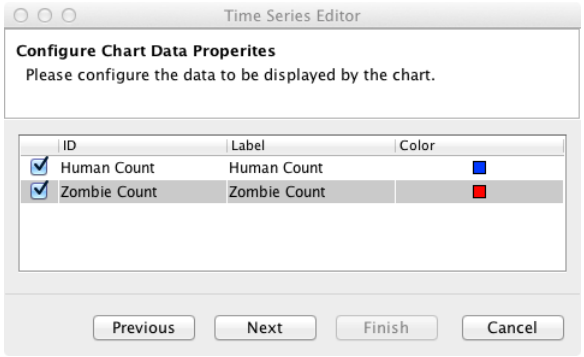

FIGURE 19. Chart Data Selection

1.2. **Model Parameters.** Recall that we hard coded the initial number of zombies and humans in our JZombiesBuilder. We shall now convert these to *model parameters*. A model parameter is parameter used by the model that a user can set via the GUI. We can create these in the Repast Simhony Runtime. Launch the JZombies model from eclipse. Use the tabs on the bottom left side of the GUI (where the Scenario Tree is) to select Parameters panel. Click the "Add Parameter" button (fig. 20) at the top of the parameters panel. This will bring up the "Add Parameter" dialog.

The "Add Parameter" dialog has the following parts.

- Name - A unique identifying name for this parameter. Each model parameter should have a unique name.

FIGURE 20. Parameters Panel

- Display Name - The label that will be used in the parameters panel for this model parameter. This does not have to be unique.
- Type - This can be an int, long, double, or string. You can also add the fully qualified name of any other type.
- Default Value - The initial value of the parameter.
- Converter - [Optional] The name of a class that extends repast.simphony.parameter.StringConverter. A StringConverter can convert non-standard types to and from a String. This is not necessary unless you use a type other than int, long, double, or string.
- Values - [Optional] A space separated list of values of the chosen type. The parameter will be restricted to these values.

To add our model parameters,

(1) Type zombie_count for the Name.
(2) Type Zombie Count for Display Name.
(3) Type int for the Type.
(4) Type 5 for the Default Value.
(5) Click OK.
(6) Click the "Add Parameter" button (fig. 20) again.
(7) Type human_count for the Name.
(8) Type Human Count for Display Name.
(9) Type int for the Type.
(10) Type 200 for the Default Value.
(11) Click OK.

The parameters panel should now look like fig. 21.

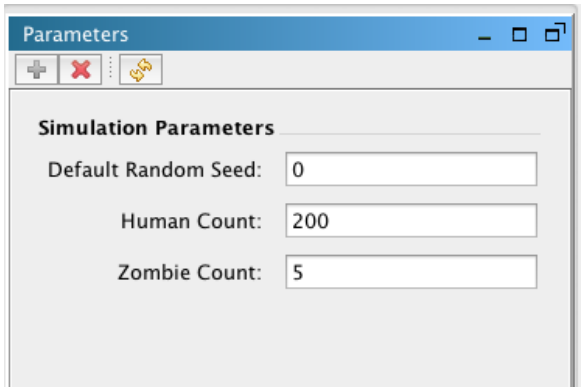

FIGURE 21. Parameters Panel With Model Parameters

To use our parameters in our model we need to retrieve them. We can use the `RunEnvironment` singleton[11] to get access to the `Parameters` object. We can then get the parameter's value using `getValue(String paramName)`. Note that the parameter's value is returned as an Object so we need to cast it appropriately. Make the following changes to JZombiesBuilder. Replace the hard coded `zombieCount` value with:

```
1 // correct import: repast.simphony.parameters.Parameters
2 Parameters params = RunEnvironment.getInstance().getParameters();
3 int zombieCount = (Integer)params.getValue("zombie_count");
```

and the `humanCount` with:

```
1 int humanCount = (Integer)params.getValue("human_count");
```

Lauch the jzombies Model, set the parameters and run. Note that the parameters will reset to their default values when the simulation is reset.

1.2.1. *Stochastic Execution and Parameter Sweeps.* Most Repast models use random draws and are therefore stochastic simulations. Stochastic simulations will produce different outcomes for different random number streams, which are generally driven by choosing different random seeds. Such simulations should be executed repeatedly to explore the space of possible outcomes. Even without randomness, model sensitivity analysis parameter sweeps usually should be run to determine the response of the model to changes in input values. Repast provides both local and distributed tools for automatically completing

[11]See RunEnvironment in the Java API doc for more info.

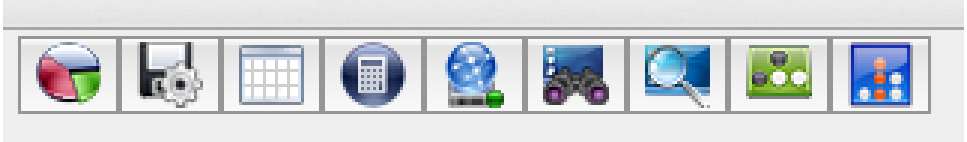

FIGURE 22. The External Tools buttons in the Repast Simphony runtime.

stochastic execution runs and parameter sweeps. Please see the Repast Simphony Batch Runs Getting Started Document for more info.

1.3. **External Tools.** We can also directly connect to some external analysis tools (fig. 22). Some of the tools require you, the user, to download additional software and are licensed differently than Repast Simphony. Three of the tools are internal to Repast Simhony and should work out of the box. If you mouse over the buttons themselves, you will see the name of the analysis tool the button refers to. The following analysis tool integrations are available.

- RStudio Statistical Computing Application
- Table of Agents and their properties
- Spreadsheet (Excel by default)
- JUNG (Internal tools that provides some stats on networks)
- Export a Geography Layer to a Shapefile
- Weka Data Mining Application
- Pajek Network Analysis Application
- JoSQL (Runs SQL like queries on simulation components – contexts etc.)

We will now experiment with integrating with Excel. Launch the jzombies model again Initialize and run the model for a few hundred ticks. Click on the Spreadsheets tool plugin button, (the button with the calculator icon) and then Next. If you have Excel on your machine, chances are the default location is correct. Otherwise select the appropriate location via the Browse button. Click Next and you should see that the File Sink 'Count Output' is selected (Fig. 23). Click Finish and Excel should launch with the data displayed in a spreadsheet. We recommend experimenting with the various other external tool plugins on your own.

1.4. **Model Distribution.** Repast models can be distributed to model users via the installation builder. This feature packs up your model and all of the software you need to run it, except for a properly configured Java Runtime Environment, into a single Java archive ("JAR") file that can be given to model users. The resulting installer can be executed on any system with a Java version equal to or greater than the version you used to compile

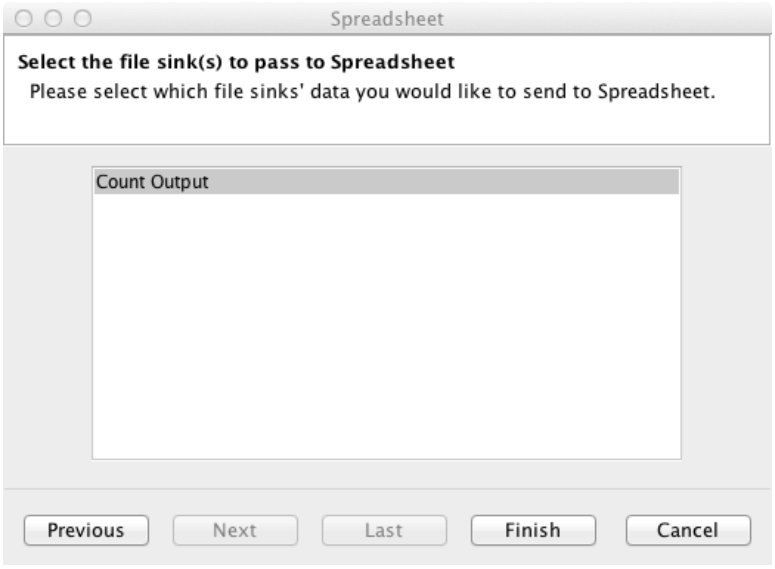

FIGURE 23. The Spreadsheet wizard with the File Sink selected.

the model. For example, if you used Java 1.8 to compile the model, then Java version 1.8 or greater must be used to run the installed model. If you used 1.7, then 1.7 or 1.8 can be used. Users simply copy the installer file onto their Windows, Mac OS, or Linux computers and the start the installer by double clicking on the file. Once the installer is started it will show an installation wizard that will prompt the user for the information needed to install the model. If desired, the installer can also be run in a command line mode.

Building an installer for a model is straightforward. In Eclipse, simply choose the "Build Installer for ⟨Your Model Name Here⟩ Model" and provide a location and name for the installer file. The installer file's default name is "setup.jar," which is suitable for most purposes. The install builder will then package and compress your model and the supporting Repast software. The resulting installer files are about 70 MB plus the size of the model code and data. 75 MB to 80 MB is a common total size.

The Repast install builder uses the IzPack system. More information on installer customization and use, including command line activation, can be found on the IzPack web site.